Elastic Pipelines and Basics of Multi-rule Systems

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

February 16, 2011

Elastic pipeline
Use FIFOs instead of pipeline registers

Can all three rules fire concurrently?

\[\text{rule stage1 (True); \ }
\text{fifo1.enq(f1(inQ.first()); \ }
\text{inQ.deq(); \ endrule \ }
\text{rule stage2 (True); \ }
\text{fifo2.enq(f2(fifo1.first()); \ }
\text{fifo1.deq(); \ endrule \ }
\text{rule stage3 (True); \ }
\text{outQ.enq(f3(fifo2.first()); \ }
\text{fifo2.deq(); \ endrule} \]
Concurrency analysis and rule scheduling

Guarded Atomic Actions (GAA): Execution model

Repeatedly:
- Select a rule to execute
- Compute the state updates
- Make the state updates

Highly non-deterministic
User annotations can help in rule selection

Implementation concern: Schedule multiple rules concurrently without violating one-rule-at-a-time semantics
some insight into
Concurrent rule firing

- There are more intermediate states in the rule semantics (a state after each rule step)
- In the HW, states change only at clock edges

Parallel execution reorders reads and writes

- In the rule semantics, each rule sees (reads) the effects (writes) of previous rules
- In the HW, rules only see the effects from previous clocks, and only affect subsequent clocks
Correctness

Rules are allowed to fire in parallel only if the net state change is equivalent to sequential rule execution.

Consequence: the HW can never reach a state unexpected in the rule semantics.

A compiler can determine if two rules can be executed in parallel without violating the one-rule-at-a-time semantics.

James Hoe, Ph.D., 2000
Rule: As a State Transformer

A rule may be decomposed into two parts \(\pi(s)\) and \(\delta(s)\) such that

\[
s_{next} = \text{if } \pi(s) \text{ then } \delta(s) \text{ else } s
\]

\(\pi(s)\) is the condition (predicate) of the rule, a.k.a. the “CAN_FIRE” signal of the rule. \(\pi\) is a conjunction of explicit and implicit conditions.

\(\delta(s)\) is the “state transformation” function, i.e., computes the next-state values from the current state values.

Executing Multiple Rules Per Cycle:

Conflict-free rules

rule ra (z > 10);
  x <= x + 1;
endrule

rule rb (z > 20);
  y <= y + 2;
endrule

Rule \(a\) and Rule \(b\) are conflict-free if

\[
\forall s. \pi_a(s) \land \pi_b(s) \Rightarrow 1. \pi_a(\delta_b(s)) \land \pi_b(\delta_a(s)) \land 2. \delta_a(\delta_b(s)) = \delta_b(\delta_a(s))
\]

Parallel execution can also be understood in terms of a composite rule.

Parallel execution behaves like \(ra < rb\) or equivalently \(rb < ra\).

rule ra_rb;
  if (z>10) then x <= x+1;
  if (z>20) then y <= y+2;
endrule
Mutually Exclusive Rules

- Rule_a and Rule_b are mutually exclusive if they can never be enabled simultaneously

\[ \forall s . \pi_a(s) \Rightarrow \neg \pi_b(s) \]

Mutually-exclusive rules are Conflict-free by definition

Executing Multiple Rules Per Cycle:

**Sequentially Composable rules**

```
rule ra (z > 10);
x <= y + 1;
endrule

rule rb (z > 20);
y <= y + 2;
endrule
```

Rule_a and Rule_b are sequentially composable if

\[ \forall s . \pi_a(s) \land \pi_b(s) \Rightarrow 1. \pi_b(\delta_a(s)) \]
\[ 2. \text{Prj}_{R(rb)}(\delta_b(s)) = \text{Prj}_{R(rb)}(\delta_b(\delta_a(s))) \]

Parallel Execution can also be understood in terms of a composite rule

```
rule ra_rb;
if (z>10) then x <= y+1;
if (z>20) then y <= y+2;
endrule
```
Compiler determines if two rules can be executed in parallel

**Rule**a and **Rule**b are conflict-free if
\[ \forall s. \pi_a(s) \land \pi_b(s) \Rightarrow \begin{cases} \pi_b(\delta_a(s)) \land \pi_a(\delta_b(s)) \\ \delta_a(\delta_b(s)) = \delta_b(\delta_a(s)) \end{cases} \]

**Rule**a and **Rule**b are sequentially composable if
\[ \forall s. \pi_a(s) \land \pi_b(s) \Rightarrow \begin{cases} \pi_b(\delta_a(s)) \\ \delta_b(\delta_a(s)) = \delta_a(\delta_b(s)) \end{cases} \]

These conditions are sufficient but not necessary. These properties can be determined by examining the domains and ranges of the rules in a pairwise manner.

Parallel execution of CF and SC rules does not increase the critical path delay

Conflicting rules

```plaintext
rule ra (True);
  x <= y + 1;
endrule

rule rb (True);
  y <= x + 2;
endrule
```

Assume \( x \) and \( y \) are initially zero.

Concurrent execution of these can produce \( x=1 \) and \( y=2 \) but these values cannot be produced by any sequential execution:
- ra followed by rb would produce \( x=1 \) and \( y=3 \)
- rb followed by ra would produce \( x=3 \) and \( y=2 \)

Such rules must be executed one-by-one and not concurrently
The compiler issue

- Can the compiler detect all the conflicting conditions?  
  - Important for correctness  
  - Yes

- Does the compiler detect conflicts that do not exist in reality?  
  - False positives lower the performance  
  - Yes
  - The main reason is that sometimes the compiler cannot detect under what conditions the two rules are mutually exclusive or conflict free

- What can the user specify easily?  
  - Rule priorities to resolve nondeterministic choice

In many situations the correctness of the design is not enough; the design is not done unless the performance goals are met

Concurrency in Elastic pipeline

Consider rules stage1 and stage2:

```plaintext
rule stage1 (True);
  fifo1.enq(f1(inQ.first()));
inQ.deq();  
endrule

rule stage2 (True);
  fifo2.enq(f2(fifo1.first()));
fifo1.deq();  
endrule

rule stage3 (True);
  outQ.enq(f3(fifo2.first()));
fifo2.deq();  
endrule
```

Can all three rules fire concurrently?
Concurrency in FIFOs

One-Element FIFO

```verilog
module mkFIFO1 (FIFO#(t));
    Reg#(t)    data  <- mkRegU();
    Reg#(Bool) full  <- mkReg(False);
    method Action enq(t x) if (!full);
        full <= True;     data <= x;
    endmethod
    method Action deq() if (full);
        full <= False;
    endmethod
    method t first() if (full);
        return (data);
    endmethod
    method Action clear();
        full <= False;
    endmethod
endmodule
```
Two-Element FIFO

d1 d0

Assume, if there is only one element in the FIFO it resides in d0

Two-Element FIFO

module mkFIFO (FIFO#(t));
Reg#(t) d0  <- mkRegU();
Reg#(t) d1  <- mkRegU();
Reg#(Bool) v0 <- mkReg(False);
Reg#(Bool) v1 <- mkReg(False);
method Action enq(t x) if (!v1);
if v0 then begin d1 <= x; v1 <= True; end
else begin d0 <= x; v0 <= True; end endmethod
method Action deq() if (v0);
if v1 then begin d0 <= d1; v1 <= False; end
else begin v0 <= False; end endmethod
method t first() if (v0);
return d0; endmethod
endmodule

Assume, if there is only one element in the FIFO it resides in d0

Two-Element FIFO

method Action enq(t x) if (!v1);
if v0 then begin d1 <= x; v1 <= True; end
else begin d0 <= x; v0 <= True; end endmethod
method Action deq() if (v0);
if v1 then begin d0 <= d1; v1 <= False; end
else begin v0 <= False; end endmethod
endmodule

Assume, if there is only one element in the FIFO it resides in d0

Two-Element FIFO Analysis

method Action enq(t x) if (!v1);
if v0 then begin d1 <= x; v1 <= True; end
else begin d0 <= x; v0 <= True; end endmethod
method Action deq() if (v0);
if v1 then begin d0 <= d1; v1 <= False; end
else begin v0 <= False; end endmethod

Turn methods into rules for analysis

rule enq if (!v1);
if v0 then begin d1 <= x; v1 <= True; end
else begin d0 <= x; v0 <= True; end endrule
rule deq if (v0);
if v1 then begin d0 <= d1; v1 <= False; end
else begin v0 <= False; end endrule

Do rules enq and deq conflict?
Two-Element FIFO
Analysis cont.

rule enq if (!v1);
if v0 then begin d1 <= x; v1 <= True; end
else begin d0 <= x; v0 <= True; end endrule

rule deq if (v0);
if v1 then begin d0 <= d1; v1 <= False; end
else begin v0 <= False; end endrule

Split rules for analysis

What represents the possibility of simultaneous enq and deq?

rule enq1 if (!v1 && v0);
d1 <= x; v1 <= True; endrule
rule enq2 if (!v1 && !v0);
d0 <= x; v0 <= True; endrule
rule deq1 if (v0 && v1);
d0 <= d1; v1 <= False; endrule
rule deq2 if (v0 && !v1);
v0 <= False; endrule

Two-Element FIFO
a "more optimized" version

module mkFIFO (FIFO#(t));
  Reg#(t)    d0  <- mkRegU();
  Reg#(Bool) v0  <- mkReg(False);
  Reg#(t)    d1  <- mkRegU();
  Reg#(Bool) v1  <- mkReg(False);
method Action enq(t x) if (!v1);
  v0 <= True; v1 <= v0;
  if v0 then d1 <= x; else d0 <= x; endmethod
method Action deq() if (v0);
  v1 <= False; v0 <= v1; d0 <= d1; endmethod
method t first() if (v0);
  return d0; endmethod
method Action clear();
  v0<= False; v1 <= False; endmethod
endmodule

Assume, if there is only one element in the FIFO it resides in d0
How can we express designs with such concurrency properties reliably?

RWire to the rescue

```
interface RWire#(type t);
    method Action wset(t x);
    method Maybe#(t) wget();
endinterface
```

Like a register in that you can read and write it but unlike a register
- read happens after write and is Valid only if a write occurs in the same cycle
- data disappears in the next cycle
One-Element Pipeline FIFO

module mkPipelineFIFO1 (FIFO#(t));
  Reg#(t) data  <- mkRegU();
  Reg#(Bool) full  <- mkReg(False);
  RWire#(void) deqEN <- mkRWire();
  Bool deqp = isValid (deqEN.wget());
method Action enq(t x) if (!full || deqp);
  full <= True;     data <= x;
endmethod
method Action deq() if (full);
  full <= False; deqEN.wset(?);
endmethod
method t first() if (full);
  return (data);
endmethod
method Action clear();
  full <= False;
endmethod endmodule

This works correctly in both cases (fifo full and fifo empty)

Rwire allows us to create a combinational path between enq and deq but does not affect the conflict analysis

Conflict analysis: Rwire deqEN allows concurrent execution of enq & deq with the functionality deq<enq; However, the conflicts around Register full remain!
Solution - Config registers

**Lie a little**

- **ConfigReg is a Register (Reg#(a))**
  
  ```
  Reg#(t) full <- mkConfigRegU;
  ```

- **Same HW as Register, but the definition says read and write can happen in either order**
  - However, just like a HW register, a read after a write gets the old value

- **Primarily used to fool the compiler analysis to do the right thing**

---

One-Element Pipeline FIFO

**A correct solution**

```
module mkLFIFO1 (FIFO#(t));
  Reg#(t) data  <- mkRegU();
  Reg#(Bool) full <- mkConfigReg(False);
  RWire#(void) deqEN <- mkRWire();
  Bool deqp = isValid (deqEN.wget()));

  method Action enq(t x) if (!full || deqp);
    full <= True; data <= x;
  endmethod

  method Action deq() if (full);
    full <= False; deqEN.wset(?);
  endmethod

module mkLFIFO2 (FIFO#(t));
  Reg#(t) data  <- mkRegU();
  Reg#(Bool) full <- mkConfigReg(False);
  RWire#(void) deqEN <- mkRWire();
  Bool deqp = isValid (deqEN.wget()));

  method Action enq(t x) if (!full || deqp);
    full <= True; data <= x;
  endmethod

  method Action deq() if (full);
    full <= False; deqEN.wset(?);
  endmethod
```

---

---

No conflicts around full:
when both enq and deq happen; if we want deq < enq then full must be set to True in case enq occurs.

Scheduling constraint on deqEn forces deq < enq

- first < enq
- deq < enq
- enq < clear
- deq < clear
FIFOs

- Ordinary one element FIFO
  - deq & enq conflict
- Pipeline FIFO
  - first < deq < enq < clear
- Bypass FIFO
  - enq < first < deq < clear

All in the BSV library

---

An aside

Unsafe modules

- Bluespec allows you to import Verilog modules by identifying wires that correspond to methods
- Such modules can be made safe either by asserting the correct scheduling properties of the methods or by wrapping the unsafe modules in appropriate Bluespec code

Config Reg is an example of an unsafe module
Takeaway

- FIFOs with concurrent operations are quite difficult to design, though the amount of hardware involved is small
  - FIFOs with appropriate properties are in the BSV library
- Various FIFOs affect performance but not correctness
- For performance, concentrate on high-level design and then search for modules with appropriate properties

Next lecture: dead cycles